Approaches to Legacy System Evolution

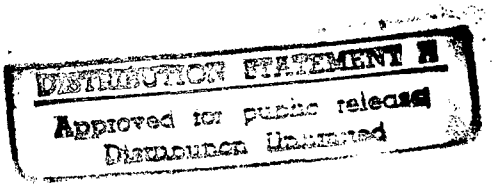
Nelson H. Weiderman

John K. Bergey

Dennis B. Smith

Scott R. Tilley

December 1997



TECHNICAL REPORT
CMU/SEI-97-TR-014
ESC-TR-97-014

Title :Approaches to Legacy System Evolution

Technical Report
CMU/SEI-97-TR-014
ESC-TR-97-014
December 1997

Approaches to Legacy System Evolution

Nelson H. Weiderman

John K. Bergey

Dennis B. Smith

Scott R. Tilley

Reengineering Center
Product Line Systems

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Table of Contents

List of Figures

Acknowledgements

Linda Northrop and Paul Clements of the Software Engineering Institute, and Steve Woods of the University of Hawaii at Manoa, provided valuable comments on early drafts of this report.

Approaches to Legacy System Evolution

Abstract: The approach that one chooses to evolve software-intensive systems depends on the organization, the system, and the technology. We believe that significant progress in system architecture, system understanding, object technology, and net-centric computing make it possible to economically evolve software systems to a state in which they exhibit greater functionality and maintainability. In particular, interface technology, wrapping technology, and network technology are opening many opportunities to leverage existing software assets instead of scrapping them and starting over. But these promising technologies cannot be applied in a vacuum or without management understanding and control. There must be a framework in which to motivate the organization to understand its business opportunities, its application systems, and its road to an improved target system. This report outlines a comprehensive system evolution approach that incorporates an enterprise framework for the application of the promising technologies in the context of legacy systems.

1 Introduction

As system assets become a greater proportion of an enterprise's total assets, more attention is being focused on the processes and technology used to build and maintain those assets. Increasingly, software is being viewed as an asset that represents an investment that grows in value rather than a liability whose value depreciates over time.

When systems are small and touch only a small number of an organization's activities, it is possible to consider redesigning and replacing a system or subsystem that can no longer satisfy enterprise needs. But systems that grow to the extent that they touch many different activities become substantial investments whose replacement is more difficult. Therefore, a major issue today is how to build software assets that will provide leverage in building future software assets.

One of the reasons that the situation is changing so rapidly is the emergence of integrating infrastructures. With improved integration we have seen the World Wide Web (the Web) and electronic commerce flourish. Where once information systems were isolated and difficult to access, they can now be accessed using the Web and interfacing software.

The Internet is being used in a number of innovative ways to connect users both inside organizations and between organizations. Within organizations, intranets are not only being used to connect departments such as marketing, sales, and engineering, but also to connect teams of software developers around the world working around the clock on the same project. Between organizations, intranets are being used to connect businesses with their suppliers and their customers. It is becoming a medium for placing orders, receiving delivery, and checking status.

As a result, integration mechanisms have helped to turn software liabilities into software assets. Managers see opportunities in software assets where once they saw only trouble. Now they see how software can be leveraged both horizontally across their enterprise and vertically to their customers and suppliers.

As evidence of this software renaissance, one sees attention focused on "architecture" and "product line systems." Just as manufacturing industries have found that they can leverage their asset base by restricting the amount of variation in their products, the information systems industries are learning how to develop software to serve multiple purposes by restricting and managing the amount of variation allowed in the software. They have learned that a core set of software assets can be effectively used as leverage to produce valuable inventories of evolvable software application systems.

In traditional software development, systems are handcrafted as one-of-a-kind products. Integration between systems is nontrivial, and there is little systematic reuse of assets in other systems. On the other hand, a software product line is a set of software-intensive systems sharing a managed set of features that address a particular market segment or fulfill a particular mission. Substantial economies can be achieved when the systems are developed from a common set of organizational assets. A product line approach enables the systematic leveraging and reuse of software assets.

The leverage of product line systems means that a variety of assets can be leveraged each time a product is created from a product line. The items that can provide such leverage include requirements, architectural design, components, modeling and analyses, testing, and processes. As a result, with appropriate planning and management, there is potential for substantial cost savings.

On the other hand, unintegrated (stovepipe) software assets that are not used for continuous production of additional assets become stale and require more and more assets to maintain them. Hence their value decreases over time, and eventually there may be more cost associated with their continued maintenance than benefit from their continued use. At that point the software becomes a liability. This concept is illustrated in Figure 1.

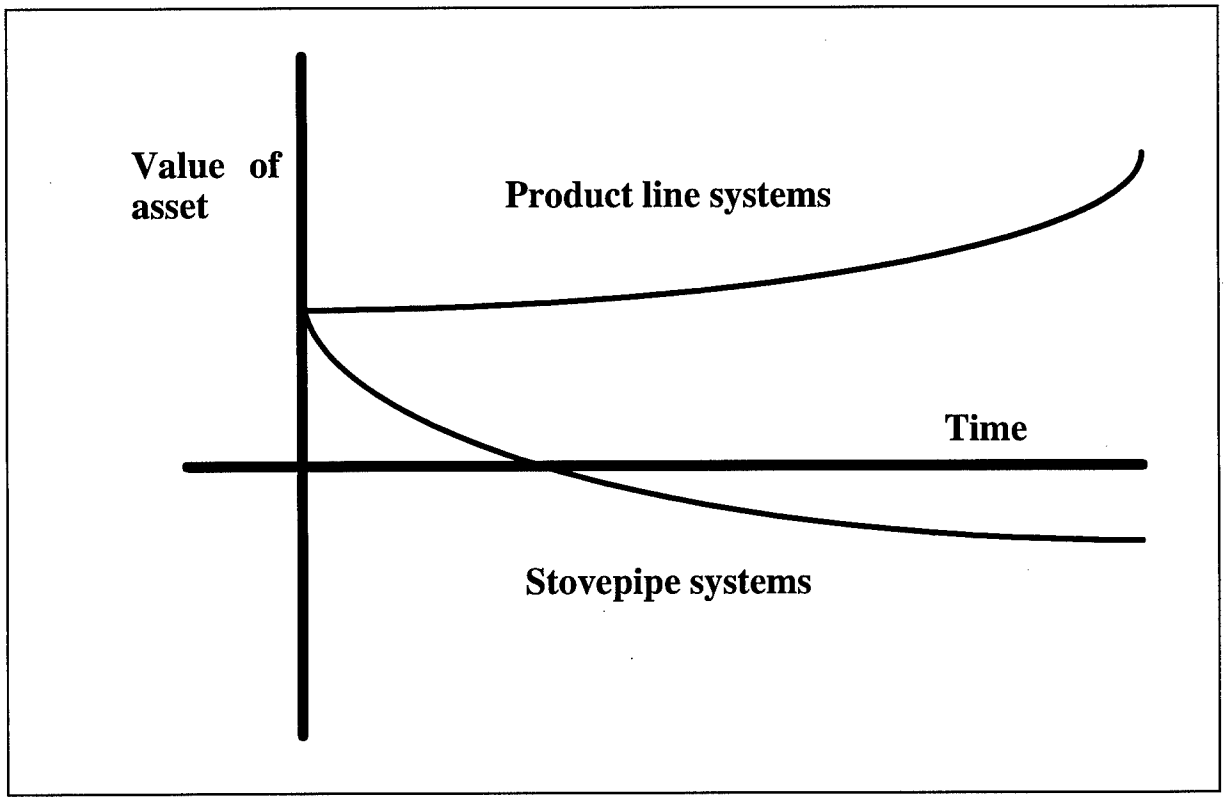


Figure 1: Conceptual Value of Software Assets

New product starts should plan for systematic reuse of software assets in advance. *A priori* development of systems using a product line practice approach is an active research area and deserves continuing attention. For example see [Bass 97]. There have been cases where a product line approach was taken out of expediency where it was known in advance that the resources were not available to construct two similar systems [Brownsword 96].

In most contexts, however, developers are not faced with a blank slate. Rather, they are faced with legacy systems (also known as "heritage applications") that have developed over many years at a substantial cost. These legacy systems represent a patchwork of mainframe, minicomputer, and desktop applications, both centralized and distributed, under dispersed control. Geography, database incompatibilities, and corporate mergers can fragment them. But still there is a requirement to maximize the information system assets by protecting, managing, integrating, and modernizing the legacy systems.

There are many ways of getting to a product line. Regardless of the starting point, the goal is to develop higher quality systems, faster, with higher productivity and improved efficiency. Product lines accomplish this goal by facilitating the systematic reuse of software assets. The emphasis is on strategic, coarse-grained reuse that leverages models, architectures, designs, documentation, testing artifacts, people, processes, and implementations.

The purpose of this report is to focus on the difficult, real-world form of leverage, namely starting with legacy system assets and evolving them toward a core set of product line assets. We posit that this leveraging of legacy assets is enabled through the convergence of a set of practices, which we try to describe at a high level in this report. The approach we use is to elaborate on each of the following:

- distinguishing between system evolution and software maintenance
- using an enterprise approach for guiding decision making for system evolution
- developing a technical understanding of systems at a high level of abstraction
- using distributed object technology and wrapping for system evolution
- using net-centric computing for system evolution

We also present credible evidence of progress and experience that supports this approach. While the range of application for these ideas may not include all classes of systems (real-time embedded applications may be one such exception), we believe that the applicable scope for this approach is quite broad.

2 Distinguishing Software Maintenance from System Evolution

We first make a distinction between software maintenance and system evolution for reengineering. We define software maintenance as a fine-grained, short-term activity focused on (perhaps a large number of) localized changes. The Year 2000 (Y2K) problem [Smith 97], in most cases, is an example of pervasive localized software maintenance. This is because dates are most often handled idiosyncratically throughout the system rather than by one single routine that is called in many places. This is precisely why the Y2K problem is so difficult and so resistant to high-level solutions. Software will always have these low-level maintenance requirements, and some of these changes can add value rather than merely maintaining the value of the software.

With software maintenance, the structure of the system remains relatively constant and the changes produce few economic and strategic benefits. There is a tendency to respond to one software requirement at a time. There are few economies of scale that accrue from software maintenance. There is little in the way of enhanced reuse. Only through structural change can the software provide leverage for further development efforts. Thus we strive to increase the asset value of software by making it capable of accepting substantive structural changes.

System evolution is a coarser grained, higher level, structural form of change that makes the software systems qualitatively easier to maintain. Evolution allows the system to comply with broad new requirements and gain whole new capabilities. Instead of changing software only at the level of instructions in a higher level programming language, change is made at the architectural level. System evolution increases the strategic and economic value of the software by making it easier to integrate with other software and making it more of an asset than a liability.

Both bottom-up and top-down approaches can lead to substantive structural change. From a bottom-up perspective, it is possible to start with a detailed code review, and build up a new structure and a new form of documentation so that the system is qualitatively easier to maintain [Buss 94]. Architectural extraction techniques may also start with source code analysis [Kazman 97].

A top-down approach treats the software more as black boxes that can be reformulated for integration with other systems, rather than white boxes that must be enhanced at a low level. It is becoming more sensible and economical to reengineer legacy systems by treating their components as black boxes and reinterfacing than it is to fully understand what is inside these boxes. This black-box approach is preferred because the technology for interfacing and integrating is developing much faster than the technology for program understanding. For a more detailed treatment of white-box and black-box reengineering, see [Weiderman 97]. In Section 4 we will briefly describe white-box and black-box technologies and their impact on system evolution.

It is clear that recent emphasis on architecture and product lines has shifted the focus from software maintenance to system evolution. Systems with a well-conceived architecture allow the software to interact across well-defined interfaces without regard for internal implementation details. This encapsulation permits the systems to be upgraded and enhanced at a structural level. Systems that are part of a product line family have a core set of assets that form the basis for further development and evolution of a group of similar systems. There is an increased emphasis on components that can interact easily with other components.

What has received less attention than necessary in the product line work is how to make architectural structuring and product line practices work in the context of legacy systems. The remainder of this report will address these important issues.

3 Using an Enterprise Approach for Decision Making

System evolution and technology insertion do not take place in a vacuum. Many attempts at evolution and migration fail because they concentrate on a narrow set of software issues without considering the broader set of management and technical issues. Evolution takes place in the context of an organizational setting that varies considerably in terms of the culture and the readiness to incorporate change. While there may be many complex technical problems that are largely unprecedented, a focus on the technical problems to the exclusion of the enterprise problems is a recipe for disaster. Hence it is crucial to plan for change in the context of the enterprise.

The SEI has developed an enterprise framework for the disciplined evolution of legacy systems [Bergey 97] as a guide for organizations planning software evolution efforts, such as migrating legacy systems, to more distributed open environments. This framework draws out the important global issues early in the planning cycle and provides insight and guidance for a disciplined evolution approach.

In addition to the software engineering and technology considerations, the enterprise approach addresses the needs of the customer, the organization's strategic goals and objectives, the operational context of the enterprise, as well as the current legacy systems and their operational environment. It recognizes the central importance of both software engineering and systems engineering (and their interplay) to the system evolution initiative. The seven elements of the framework are shown in Figure 2.

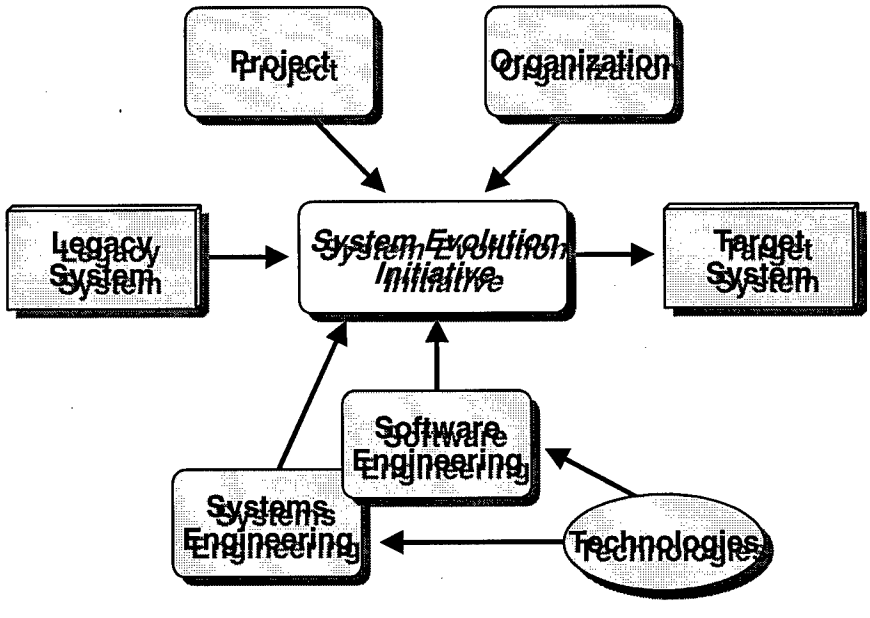


Figure 2: A Framework for the Disciplined Evolution of Legacy Systems

These elements are applicable to a wide class of system evolution initiatives. In practice, the specific composition of the framework and the interrelationships between the elements are a function of the enterprise, its culture, and its management and technical practices (life-cycle activities, processes, and work products that are used to carry out the tasks described in the project plan and migration strategy).

One of the important characteristics of the framework is its checklist of issues. The framework does not prescribe an evolution process, but rather clearly outlines the issues. The use of checklists provides insight into who the stakeholders and decision makers are, and what enterprise factors and work products govern the tasks and the decision-making processes. The checklists may also surface "gray areas" in the enterprise planning, such as how interdependent aspects of the work will be coordinated with external organizations and customers, and how the proposed system will potentially be affected by (or affect) other enterprise efforts that are already underway or in the planning stages.

As an example, the following questions are from the organization checklist:

- Has a common vision been developed and communicated?
- Have all three project variables—capability, schedule, and cost—been predetermined by the organization prior to developing a project plan?
- Has the feasibility of evolving the legacy system been predetermined? Or the benefits?

The following questions are from the project checklist:

- Will a project team of key interdisciplinary engineers be established to serve as a system design team? If not, how will global systems engineering issues and specialty engineering requirements (e.g., security) be adequately addressed and coordinated?
- Does the project plan clearly describe the system evolution strategy? Are the project team members fully supportive of the strategy?

The enterprise framework is proving to be a useful tool for probing and evaluating planned and ongoing system evolution initiatives, for drawing out important global issues early in the planning cycle using the checklists as a guide, and for providing insight for developing a synergistic set of management and technical practices. In two early uses of the framework, it was determined that the organizations' initial perception of their problems differed from the real problem. In one case the major enterprise deficiency was the lack of a global planning and coordination process and effective systems engineering infrastructure. In another case the major deficiency turned out to be the lack of a defined and repeatable process for evaluating legacy systems to serve as a framework for identifying subsidiary tasks and evaluation criteria.

The enterprise framework can help organizations leverage their assets in two ways. First, it can suggest a disciplined approach pointing to some of the prerequisite skills and infrastructure support that are required before attempting to move toward a product line approach. Second, it can provide guidance for reusing legacy systems as a starting point for developing reusable components for a future product family. Our plans call for expanding

the enterprise framework to incorporate scenarios (of hypothetical migration efforts) and experience reports of actual migration efforts and show the impact of moving to a product line approach from an enterprise perspective. This will instantiate the framework in ways that will make it more meaningful to users and easier to apply.

In Section 4 we describe the first step in any evolution or migration effort, namely how we come to understand the essential characteristics of the system undergoing change.

4 Developing High-Level System Understanding

Program understanding is the (ill-defined) deductive process of acquiring knowledge about a software artifact through analysis, abstraction, and generalization [Tilley 96a]. Clearly, program understanding is a prerequisite for software evolution. However, we assert that the nature of program understanding should change its emphasis from an understanding of the internals of software modules (white-box reengineering) to an understanding of the interfaces between software modules (black-box reengineering). This section will draw the distinction between these two forms of understanding and focus on the activities necessary for the latter. We will use *program understanding* for the former and *system understanding* for the latter.

Understanding is critical to our ability to evolve unproductive legacy assets (e.g., obsolete, overly constrained, or stagnating components) into reusable assets that can contribute to a product line approach. Legacy assets may be aging software systems that are constructed to run on various obsolescent hardware types, are programmed in obsolete languages, and suffer from the fragileness and brittleness that results from prolonged maintenance. As stovepipe software ages, the task of maintaining it becomes more complex and expensive, and the software becomes more of a liability than an asset. While bottom-up program understanding has its place, it is often the case that software and system engineers spend inordinate amounts of time trying to reproduce the system's high-level architecture from low-level source code.

Legacy code can be difficult to understand for many reasons. It may have been created using ad hoc methods and unstructured programming. It may have been maintained in crisis mode with no updates to the higher level documentation. There may be little or no conceptual integrity of its architecture and design. But every system has an architecture, even if it is not written down. This architecture and high-level understanding of the structure of the legacy system must be the focus of a system understanding effort.

Program and system understanding is a relatively immature field of research in which the terminology and focus are still evolving. Tilley and Smith [Tilley 96a] describe three promising lines of research: investigating cognitive aspects, developing support mechanisms, and maturing the practice. In Sections 4.1–4.3, we give brief summaries of how each of these lines of research should be tailored to a high-level, white-box form of understanding necessary for more rapid and cost-effective migration.

4.1 Cognitive Aspects

The cognitive aspect of program understanding is the study of problem-solving behavior of software engineers. Because the productivity of software engineers varies by more than an order of magnitude, the strategies of the successful practitioners are of great interest in producing tools and techniques that better support program understanding. Most of this research has focused on comprehension strategies of software engineers. A

comprehension strategy involves what information software engineers use to understand a software artifact and how they use that information. These studies cross the disciplines of software engineering, education, and cognitive science.

Studies show that software maintainers use a combination of strategies. They are usually a variation of top-down understanding, bottom-up understanding, iterative hypothesis refinement, or some combination of the three. Here is a summary of each [Tilley 96a]:

- Bottom up: Reconstruct the high-level design of a system, starting with the source code, through a series of chunking and concept assignment steps.
- Top-down: Begin with a preexisting notion of the functionality of the system and earmark individual components of the system responsible for specific tasks.
- Iterative refinement: Create, verify, and modify hypotheses until the entire system is explained by a consistent set of hypotheses.
- Combination: Opportunistically exploit top-down and bottom-up cues as they become available.

Clearly, for the high-level system understanding that we are advocating here, the second and third approaches or a combination of them is preferred. Starting a system understanding effort at the source code or using it as a primary technique should be considered an option for software maintenance or for building a solid foundation for a high-level approach. The cognitive aspects of system understanding should be focused on the external behavior rather than the internal behavior of components. These cognitive aspects can be supported by Web-based, computer-supported cooperative understanding (CSCU) for teams in geographically distributed collaborative efforts. Furthermore, as cognitive models are developed, the codified knowledge should take the form of systems evolution handbooks for capturing expertise that has been effective for general and specific systems evolution scenarios. Such handbooks could provide prescriptive solutions to common problems.

4.2 Developing Support Mechanisms

Computer-aided support mechanisms are needed to help software engineers in their system understanding tasks. But rather than helping software engineers to extract high-level information from low-level code, these support mechanisms should focus on extracting interface definitions from the program specifications. Reverse engineering is a process of examination and reconstruction. Its canonical activities are data gathering, knowledge management, and information exploration (including navigation, analysis, and presentation) [Tilley 96b].

Data gathering can often go astray by taking an automated brute-force approach to static and dynamic analyses starting with the source code. Constructing abstract syntax trees with a large number of fine-grained syntactic artifacts and dependencies does not scale up

easily because the algorithms are non-linear in the number of nodes of the system. For system understanding (as opposed to program understanding), reverse engineering should attack the user interfaces and the system interfaces. The best source of data for the user interface may be the user manuals, if they are up to date and accurate. The best source of information for system interfaces may be the descriptions of the APIs (application programmer interfaces). As more systems become object oriented, these APIs will become more ubiquitous and make the data gathering easier and more straightforward.

It should be noted that the explosive growth of both the Internet and the Web are making it possible for program and system understanding technology to be delivered to practitioners in a familiar form (Web-based interfaces). By using extended markup language (XML) as the basis for a common intermediate form, many of the capabilities of the Web can be leveraged for use in reengineering. Examples of such capabilities include search engines, visualization tools, and integration mechanisms [Tilley 97a].

4.3 Maturing the Practice

As we shift the focus from program understanding to system understanding, from software maintenance to system evolution and migration, and from bottom-up techniques to top-down techniques, the prospects for widespread adoption will improve. One of the impediments to greater adoption of program understanding technology has been that this technology has been focused on toy programs that do not represent real-world legacy systems (although the current interest in the Y2K has rapidly changed this situation for the better). System understanding must be integrated with the other technologies described in this report, namely the Internet, distributed object technology, and net-centric computing.

One tool that addresses the need to be able to extract information from existing system implementations and reason architecturally about this information is called Dali [Kazman 97]. Dali is an open, lightweight workbench that aids an analyst in extracting, manipulating, and interpreting architectural information. Although Dali extracts information automatically from source code, it is an interactive system that assists the user in forming and interpreting the architectural information. It relies heavily on user input to extract a description of the architecture. The researchers realized that the automated approach had been tried before and had failed. Hence Dali supports the user in defining architectural patterns and in matching those patterns to extracted information. While Dali is in an early prototype stage, it is in systems like these that we find the most promise for system understanding in producing evolving systems that can form the basis for product lines.

The research in system understanding should de-emphasize the fine-grained, bottom-up, exhaustive, computationally intensive techniques in favor of coarse-grained, top-down, targeted analysis. They must carefully study the numerous successes that have been recently documented using distributed object technology (e.g., see [Weiderman 97]). There is also room for small-scale experimentation with large-scale problems. The technology now

facilitates the integration of large-scale systems using net-centric computing, distributed object technology, and middleware as is described in Sections 5 and 6.

5 Distributed Object Technology and Wrapping

The approaches to the evolution of legacy systems are being dramatically changed by distributed object technology and wrapping. Traditionally, the approach taken for legacy systems reengineering has been to understand a system's structure and to extract its essential functionality so the whole system or a series of pieces of the system could be transformed into a more evolvable system over the long term. But distributed object technology is dramatically changing the nature and economics of legacy system reengineering in two important ways.

The first impact is the new approach to reengineering that distributed object technology offers. Traditional reengineering is based on "deep" program understanding and reverse engineering. The cost/benefit ratio of this approach is staying the same in the face of new technologies such as Common Object Request Broker Architecture (CORBA), Java, and the Web. However, the benefits of "shallow" interface understanding and component wrapping using these distributed object technologies are rising rapidly relative to the replacement cost. As a result, the economic balance is changing from traditional transformation-based reengineering to wrapper-based reengineering. This may have a significant impact on many organizations struggling with updating their systems.

The second impact is on the reengineering of systems that are built using distributed objects. Current research is primarily focused on using static analysis techniques to reengineer monolithic systems. This approach may not be successful when applied to distributed systems built from off-the-shelf components that are essentially black boxes. New analysis techniques may be required to reengineer such systems. For example, binary reverse engineering, interface behavior probing, and protocol analysis may be more useful in understanding the nature of such a system.

5.1 How Object Technology Provides Leverage

Object technology (OT) has made slow, but steady, progress in influencing the course of software development since it was introduced in the late 1970s. In the object-oriented model, systems are viewed as cooperating objects that encapsulate structure and behavior and that belong to hierarchically constructed classes. While the benefits of object-oriented technology have been demonstrated, and most new systems are being constructed using its principles, the transition from more traditional structured approaches can certainly not be considered complete, nor has OT been fully exploited, especially in legacy systems that predated objects. But OT is maturing rapidly and is well accepted as a technology that addresses complexity, improves maintainability, promotes reuse, and reduces life-cycle costs of software.

Objects increase leverage because of encapsulation and information hiding. They expose the interfaces of software modules while hiding their implementation details, so that they are more easily employed as reusable components. Objects exist at many levels of abstraction.

At fine-grained levels, objects can be used to create data structures such as lists, trees, and graphs that have known behaviors. At mid-grained levels, objects represent more sophisticated functions with APIs for using them. At coarse-grained levels, objects can take the form of servers which are accessed by clients who know only the system interface. These client/server systems often are used for hiding the details of accessing a database. Object-oriented systems provide leverage by enabling reuse. While early attempts at capitalizing on the leverage of OT focused on fine-grained reuse, more recent efforts have found that the real leverage comes from reuse at the architectural coarse-grained level of systems and subsystems.

5.2 How Distributed Object Computing Provides Leverage

As object technology became more popular through the 1980s, there was more interest in bundling the concept of objects with the concept of transparent distributed computing. Objects, with their inherent combination of data and behavior and their strict separation of interface from implementation, offer an ideal package for distributing data and processes to end-user applications. Objects became an enabling technology for distributed processing. Laddaga and Veitch [Laddaga 97] address the leverage of distributed objects for legacy systems by saying they are "specifically designed to support rapidly changing software with cost proportional to that of the change, rather than the size of the entire application."

Distributed object computing (DOC) is the application of OT to distributed environments, i.e., multiple autonomous computers that are connected by a network and have no shared physical memory. The advent of smaller, more powerful, and less expensive computing engines has precipitated an interest in moving applications from mainframes and minicomputers to PCs and workstations, and in distributing functionality over multiple communicating computers. Over time, the notion of distribution has generally migrated from tightly coupled, geographically close, homogeneous machines to more loosely coupled, geographically remote, heterogeneous machines.

Early distributed systems employed a client/server model where computations on one machine (the client) invoke computations on another machine (the server) in a manner often viewed as a remote procedure call (RPC). The server process is a provider of services; the client is a consumer of services. The client/server model is concerned primarily with the problems of distributing function across local and wide area networks through devices such as pipes and sockets. The client/server model has evolved from one in which the client was a terminal accessing a server (usually a mainframe) in its first generation, to an intelligent "fat client" workstation accessing a server with less capability in its second generation, to a "three-tier" model as show in Figure 3. The three-tier model has clients that can be relatively thin in tier 1 communicating with a Web server and business objects in tier 2, which in turn communicate with legacy assets in tier 3. There are many variations on this theme, but the middle tier introduces the distributed object layer.

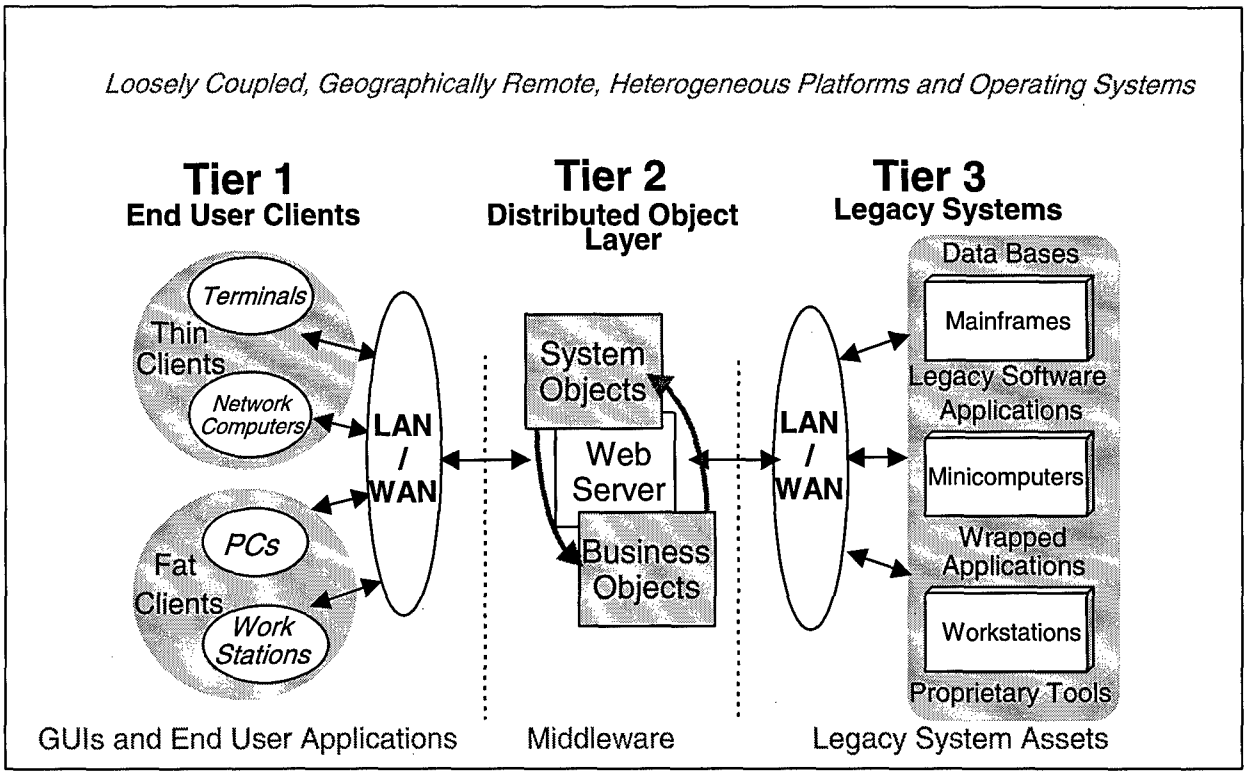


Figure 3: The Distributed Object Computing Paradigm

DOC provides additional leverage over OT by making objects available everywhere — across heterogeneous platforms and networks. No longer must two programs sharing an object be on the same system or part of a closely coupled client/server system. They may be on separate machines, running different operating systems, in different parts of the world. Objects are easier to share not only within a project, but within a division, an enterprise, and across enterprises. Enterprises can make their objects available to their employees, their customers, their suppliers, and in all other business relationships. In short, the ability to have objects everywhere creates tremendous leverage for both the enterprise and software developers.

Software architecture and product lines are concerned with design abstractions for system-level structure. By *system-level* we mean something larger than a single computer program. The significance of making distribution extensions to OT is that software designers and maintainers have at their disposal the means of expressing abstract system designs and, more importantly, have tools for quickly fabricating working versions of these designs. That is, there is a more direct path now than ever before from abstract architectural concepts to concrete implementation of these concepts using DOC.

5.3 Using Wrapping and Middleware as Integration Mechanisms

Most DOC projects involve the use of middleware technology. Middleware is defined as "connectivity software that consists of a set of enabling services that allows multiple processes running on one or more machines to interact across a network [Foreman 97]." Middleware, sometimes known as glue, facilitates integration of components (objects) in a distributed system. While the details of these technologies vary considerably, middleware products usually provide a run-time infrastructure of services used by components to interact with each other. The focus of middleware has been to make the computing environment increasingly transparent with respect to the locus of both the computing engines and the objects of computing.

Middleware can take the following forms [Foreman 97]:

- Transaction processing (TP) monitors, which provide tools and an environment for developing and deploying distributed applications.
- Remote procedure calls, which enable the logic of an application to be distributed across the network. Program logic on remote systems can be executed as simply as calling a local routine.
- Message-oriented middleware (MOM), which provides program-to-program data exchange, enabling the creation of distributed applications. MOM is analogous to email in the sense that it is asynchronous and requires the recipients of messages to interpret their meaning and to take appropriate action.
- Object Request Brokers (ORBs), which enable the objects that constitute an application to be distributed and shared across heterogeneous networks.

In the early 1990s an international trade association called the Object Management Group (OMG) defined a standard for the distribution of objects [OMG 97a]. The OMG defined CORBA, which provided a standard by which OT could be used in distributed computing environments. The latest version of this standard, CORBA 2.0, addresses issues related to interface, registration, databases, communication, and error handling. When combined with other object services defined by OMG's Object Management Architecture (OMA), CORBA becomes a middleware that facilitates full exploitation of object technology in a distributed system.

CORBA is concerned with interfaces and does not specify implementation; CORBA is a standard for which there are many (more than a dozen at present) current products referred to as ORBs. The OMG sponsors a permanent showcase on the Web to demonstrate the interoperability between ORBs from various vendors according to the CORBA 2.0 specification [OMG 97b]. The major ORB product that is not compliant with CORBA is Microsoft's Distributed Component Object Model (DCOM). In addition to CORBA product offerings, we are now seeing product sets offering complete integration solutions for a wide range of fragmented legacy applications [Bracho 97].

There are many ways that middleware can be employed for wrapping. One way of classifying wrapping is to relate it to the nature of the mismatches (underlying assumptions) that are being addressed by the wrapping. Middleware can address mismatches of the underlying environment (hardware, operating system, programming language), mismatches in the underlying database model, mismatches in the functional requirements, and mismatches in how components interact with one another. At the Software Engineering Institute attention has been focused on exploring repair strategies centered on the coordination class. Summaries of three example prototype wrapping exercises using CORBA are contained in [Wallnau 97].

5.4 Examples of Successful Legacy Evolution

One example of a successful legacy evolution is Wells Fargo Bank's online electronic banking system. Wells Fargo started offering real-time access to account balances via the Web starting in May 1995 and has since expanded those services to include transferring funds, seeing cleared checks, examining credit card charges and payments, downloading transaction files, requesting service transactions, and paying bills [Wells Fargo 97]. The system has 100,000 enrolled customers and was handling 200,000 business object invocations per day as of early 1997 [Townsend 97].

Wells Fargo has accomplished this by leaving their legacy systems largely untouched while adding CORBA middleware to create a three-tiered client server system. The "customer" object and the "account" object allow the definition of a customer relationship whereby the client can first get all information about the customer's relationship with the bank and then, for each account owned by the customer, get the relevant summary information. Wells Fargo found that the key to enabling reuse of legacy systems was in having, maintaining, and sharing a well-architected enterprise object model.

The centerpiece of each monthly issue *Distributed Object Computing* magazine is a deployed case study such as Wells Fargo. They describe the development, the business case for building and deploying the application, the laundry list of technology used on the project and, when available, the staffing and deployment information. In their first eight issues, they have featured a Web-based banking system, an airline reservation system, a criminal justice suspect index system, a newsmedia system to provide personalized news and digital content, a 911 emergency response system, a management information system for monitoring and directing large projects, an electric power exchange system for the electric power industry, and an information system for a utility company. In most cases these development efforts made heavy use of legacy software assets.

As an example of the leverage provided by technologies such as CORBA and Java, Allied Signal Engines, a business unit of Allied Aerospace, has reported cost savings of $750,000 per new application [Gill 97]. This was accomplished by moving to component-based software architecture and by outsourcing a major portion of the actual coding effort to an offshore development company in India. Their wide-ranging development efforts were made

possible, at least in part, by the network-centric computing models that will be described in Section 6.

6 Net-Centric Computing

As is evident from the examples cited in the previous section, the Net[1] is causing a fundamental change in both the nature of enterprise applications and the development methods used to create them. The Net, and in particular the Web, is evolving into a ubiquitous, distributed, platform-independent, peer to peer, collaborative computing platform for software applications. The concept of "the network is the computer" is based on the premise that the Net provides a combination virtual disk drive for storage, powerful processors for computation, and executable content that permits the global execution of software applications irrespective of their physical location, their implementation language, or their operating system. The value of the Net to software engineering must be recognized and exploited. The promise of product line practice will not be fully achieved unless this medium is used to its fullest capabilities.

6.1 What Is Net-Centric Computing?

The explosion of interest in the Web has given rise to many new developments. One of them is net-centric computing (NCC). The underlying principle behind NCC is a distributed environment where applications and data are downloaded from network servers on an as-needed basis [Tilley 97b]. This is in stark contrast to the current use of powerful personal computers (PCs) that rely primarily on local resources. In some respects, NCC resembles an earlier computing era of mainframes and dumb terminals. However, there are important differences. NCC relies on portable applications than run on multiple architectures (write once, run anywhere), high bandwidth (for downloading applications on demand), and low-cost "thin clients."

Thin clients come in a variety of guises. Some may be traditional display-based terminals with no local processing. Examples of these very thin clients include X stations and the so-called Windows Terminal. Another type of thin client is a Java-based network computer (NC) that supports local processing. A third class of thin client is the Windows-based NetPC, which may also support Java. The proliferation of plug-ins that enable one type of client to access or emulate another, such as running a Windows NT session inside a Java applet on a NC, makes the distinction between different types of thin clients somewhat problematic at times.

The influence of NCC on software evolution can be summarized in three words: universality, ubiquity, and accessibility. Universality is provided by portable executable content, such as Java applets, which runs on multiple platforms and operating systems. Making established user interfaces, such as Web browsers, available on almost any client provides ubiquity.

[1] The term *Net* as used here includes the Internet (the global computer network), intranets (local networks that are usually isolated from the Internet by a firewall), and extranets (extensions of an intranet into the Internet in a secure manner).

Making vast quantities of corporate data, which previously were inaccessible in mainframe-based databases, available on the Net to the ubiquitous client software, provides accessibility.

6.2 How Net-Centric Computing Provides Leverage

One of the primary drivers of NCC is economics. Because applications and data are downloaded from servers on demand, there is a potential reduction in the cost and complexity for system administrators in managing a corporate network. Maintenance can be done at one central location rather than at thousands of sites in the organization, thereby reducing total cost of ownership (TCO). The tradeoff is that end users lose control of the ability to customize their local machines. However, they may gain significantly by increasing their productivity in their primary tasks by not being responsible for application installation, system administration, and troubleshooting tasks. Thus, NCC leverages system administration resources.

NCC can also leverage software assets in a number of ways by making them available over the Net. Since legacy application code is normally platform dependent, one must choose whether to use white-box reengineering techniques such as reverse engineering to redesign and rewrite the system, or to use black-box reengineering techniques such as DOT and wrapping. Reverse engineering can often be prohibitively expensive, so a variety of wrapping schemes may be used, as described in the following paragraphs.

In the simplest form, just the user interface might be changed. Instead of accessing the enterprise database through an idiosyncratic user interface, the database can be accessed through a Web browser. This transition has been accomplished many times by many organizations and, by now, should no longer be considered a high-risk, unprecedented form of software evolution. Many enterprise-wide intranets have taken this approach without significantly changing the underlying software base.

The next level of complexity involves partitioning the application into separate components so that the new version of the system can operate in a traditional client/server manner. Once this step has been taken, the software is better able to evolve the individual parts into a reusable set of components that can provide the basis for a product line. In the first case (changing the interface), the business benefits from the universality, ubiquity, and accessibility, but does not reap the rewards of business process reengineering (BPR). In the second case (partitioning), the leveraging of assets and return on investment becomes paramount.

In the third level of complexity, the application is restructured from a two-tiered client/server application into an *n*-tiered set of interacting components. Each component may reside on a client, a server, or it may migrate between them (in effect, a component serves multiple roles). Data may also be made similarly mobile. This arrangement provides the greatest flexibility for future evolution and gains maximum leverage from the NCC environment.

Finally, NCC can leverage legacy data in several ways. For example, through the Java database connectivity specification (JDBC), Java applications can access SQL databases in a manner similar to other applications that use object database connectivity (ODBC) services. Java applications can also access non-Java programs through a variety of client software. There are various tradeoffs between thinner and fatter clients. Thinner clients work better for users who perform repetitive tasks, for mobile knowledge workers sharing desktops, for remote workers who are difficult to support, for situations where security is important, and for replacing aging text-based terminals. Fatter clients work better for developers who rely on local processing power and where central administration would not be welcomed or appropriate. The trend seems to be toward thinner clients. They have been installed by the thousands in many information systems applications such as banks, retail stores, telemarketing applications, and overnight delivery applications. The use of thinner clients often must be accompanied by upgrades to the network and the servers that are being used.

6.3 Examples of Successful Use of Net-Centric Computing

Examples of the use of NCC technology to evolve legacy systems are abundant. The typical approach taken is to evolve the legacy system into a Web-centric computing (WCC) application (a "weblication"). WCC is a subset of NCC where a Web browser is used as the primary means of accessing legacy applications and data. In contrast, NCC applications can make use of other types of thin clients as well, as discussed above.

In a series of studies, International Data Corporation has conducted four in-depth economic analyses at major corporations. These well-documented studies show a return on investment averaging well over 1000 percent [Campbell 96]. They emphasized three themes: rapid deployment on heterogeneous platforms, widespread acceptance and use due to the ease of using the browser technology, and the realization of the promise of openness and the ability to replace components at will. In all cases, the use of intranets enabled the companies to take advantage of new enterprise strategies. The studies were conducted at Cadence Design Systems, Inc.; Booz, Allen & Hamilton; Silicon Graphics, Inc.; and Amdahl Corporation.

7 Summary and Conclusions

The approaches to software evolution are changing rapidly along with the changing technology. The changing technology is pushing the evolution of systems in several ways. Two approaches to software evolution appear to be on the decline. First, it is rarely possible, because of huge investments in legacy systems that have evolved over many years, to replace those systems and start from scratch. So the "big bang" approach to software migration is not often feasible. Second, it is increasingly less attractive to continue maintaining traditional (functional) legacy systems at the lowest level of abstraction, expecting them to evolve into maintainable assets. So the fine-grained maintenance approach is also undesirable because it neither adds value to the asset nor provides for future leverage.

The recommended new approach for systems evolution has been described in the previous sections and can be summarized as follows:

- Understand the goals and resources of the enterprise with respect to a system evolution project. Use a software evolution framework to plan a disciplined system evolution.
- Understand the legacy system at a high level of abstraction using system understanding technology, paying particular attention to interfaces and abstractions. Find the encapsulatable components of the legacy system on which to build.
- Consider middleware and wrapping technologies for encapsulating subsystems and creating distributed objects that form the basis for product line systems. Apply those technologies in accordance with the framework.
- Consider using the Web for expanding the scope of the legacy system and as a development tool. Capitalize on the universality, ubiquity, and access that the Web provides.
- Develop a concept of operations for the goals that you want to strive towards, and develop an incremental implementation strategy for evolving towards that goal. Having a goal and a strategy simplifies the problems associated with transitioning the new system into operational use (by breaking it down into manageable and predictable "chunks") and allows for mid-course corrections based on actual field experience and customer/user feedback.

As is so often the case in software engineering, this approach to software evolution raises the level of abstraction so that our resources are being used more effectively. Economic realities are pushing us from low-level maintenance activities to high-level transformations. A focus on architecture and product lines is facilitating large-scale reuse in construction, where before we were satisfied with small-scale reuse.

The use of these new approaches is still somewhat risky and advanced, but by no means unprecedented. They have been employed in prototypes, tested in small systems, and used to transform large systems. Useful and production-quality tools are now available. New developments are occurring at "Internet speed." Enterprises that ignore or delay the introduction of the new technologies do so at their peril.

References

[Bass 97] Bass, Leonard; Cohen, Sholom; Northrop, Linda; & Withey, James. *Product Line Practice Workshop Report* (CMU/SEI-97-TR-003, ADA 327610). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Bergey 97] Bergey, John; Northrop, Linda; & Smith, Dennis. *Enterprise Framework for the Disciplined Evolution of Legacy Systems* (CMU/SEI-97-TR-007, ADA 330880). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Bracho 97] Bracho, Rafael. *Integrating the Corporate Computing Environment with ActiveWeb.* Active Software, Inc., Santa Clara, CA. Available WWW <URL: http://www.activesw.com> (1997).

[Brownsword 96] Brownsword, Lisa & Clements, Paul. *A Case Study in Successful Product Line Development* (CMU/SEI-96-TR-016, ADA 315802). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996.

[Buss 94] Buss, E. et al. " Investigating Reverse Engineering Technologies for the CAS Program Understanding Project." *IBM Systems Journal* 33, 3 (1994): 477-500.

[Campbell 96] Campbell, Ian. *The Intranet: Slashing the Cost of Business.* Framingham, MA: International Data Corp., 1996.

[Foreman 97] Foreman, John; Brune, Kimberly; McMillan, Patricia; & Rosenstein, Robert. *Software Technology Reference Guide - A Prototype* (CMU/SEI-97-HB-001, ADA 305472). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Gill 97] Gill, Philip J. "CORBA Proves Its Value." *Object Magazine* 7, 8 (October 1997): 10-11.

[Kazman 97] Kazman, Rick & Carrière, S. Jeromy. *Playing Detective: Reconstructing Software Architecture from Available Evidence.* (CMU/SEI-97-TR-010, ADA 315653). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Laddaga 97] Laddaga, Robert & Veitch, James. "Dynamic Object Technology — Introduction." *Communications of the ACM* 40, 5 (May 1997): 38-40.

[OMG 97a] Object Management Group. *Welcome to OMG's Home Page* [online]. Available WWW <URL: http://www.omg.org/> (1997).

[OMG 97b] Object Management Group. *CORBAnet — The ORB Interoperability Showcase* [on-line]. Available WWW <URL: http://www.corba.net/> (1997).

[Smith 97] Smith, Dennis; Müller, Hausi; & Tilley, Scott. *The Year 2000 Problem: Issues and Implications* (CMU/SEI-97-TR-002, ADA 325361). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Tilley 96a] Tilley, Scott & Smith, Dennis. *Coming Attractions in Program Understanding* (CMU/SEI-96-TR-019, ADA 320731). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996.

[Tilley 96b] Tilley, Scott; Paul, Santanu.; & Smith, Dennis. "Toward a Framework for Program Understanding," 19-28. *Proceedings of the 4th Workshop on Program Comprehension.* Berlin, Germany: March 29-31, 1996. IEEE Computer Society Press, 1996.

[Tilley 97a] Tilley, Scott & Smith, Dennis. "On Using the Web as Infrastructure for Reengineering," 170-173. *Proceedings of the 5th Workshop on Program Comprehension.* Dearborn, Michigan: May 28-30, 1997. IEEE Computer Society Press, 1997.

[Tilley 97b] Tilley, Scott & Story, Margaret-Anne. *Report of the STEP '97 Workshop on Net-Centric Computing* (CMU/SEI-97-SR-016, ADA 330926). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Townsend 97] Townsend, Erik S. "Wells Fargo's 'Object Express'." *Distributed Object Computing* 1, 1 (February 1997): 18-27.

[Wallnau 97] Wallnau, Kurt; Weiderman, Nelson; Northrop, Linda. *Distributed Object Computing with CORBA and Java: Key Concepts and Implications* (CMU/SEI-97-TR-004, ADA 327035). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Weiderman 97] Weiderman, Nelson; Northrop, Linda; Smith, Dennis; Tilley, Scott; & Wallnau, Kurt. *Implications of Distributed Object Computing for Reengineering.* (CMU/SEI-97-TR-005, ADA 326945). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Wells Fargo 97] Wells Fargo Bank. Wells Fargo's WWW Homepage [online]. Available WWW <URL: http://www.wellsfargo.com> (1997).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (LEAVE BLANK)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1997	Final

4. TITLE AND SUBTITLE

Approaches to Legacy System Evolution

5. FUNDING NUMBERS

C — F19628-95-C-0003

6. AUTHOR(S)

Nelson H. Weiderman, John K. Bergey, Dennis B. Smith, Scott R. Tilley

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER

CMU/SEI-97-TR-014

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

ESC-TR-97-014

11. SUPPLEMENTARY NOTES

12.A DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified/Unlimited, DTIC, NTIS

12.B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

The approach that one chooses to evolve software-intensive systems depends on the organization, the system, and the technology. We believe that significant progress in system architecture, system understanding, object technology, and net-centric computing make it possible to economically evolve software systems to a state in which they exhibit greater functionality and maintainability. In particular, interface technology, wrapping technology, and network technology are opening many opportunities to leverage existing software assets instead of scrapping them and starting over. But these promising technologies cannot be applied in a vacuum or without management understanding and control. There must be a framework in which to motivate the organization to understand its business opportunities, its application systems, and its road to an improved target system. This report outlines a comprehensive system evolution approach that incorporates an enterprise framework for the application of the promising technologies in the context of legacy systems.

14. SUBJECT TERMS

legacy system evolution, distributed object technology, net-centric computing, program understanding, reverse engineering, reengineering

15. NUMBER OF PAGES

30

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL